Peter Lee
Computer Science UIL
State 2015

Java Topic List 2014-2015 Notes

Base Conversions and Arithmetic

1. Converting from base 2 to base 10: Memorize the value of every bit in groups. Find the sum of the values of the bits that contain 1.
$2^0 = 1,\ 2^1 = 2,\ 2^2 = 4 \rightarrow 1,\ 2,\ 4$
$2^3 = 8,\ 2^4 = 16,\ 2^5 = 32 \rightarrow 8,\ 16,\ 32$
$2^6 = 64,\ 2^7 = 128,\ 2^8 = 256 \rightarrow 64,\ 128,\ 256$
$2^9 = 512,\ 2^{10} = 1024,\ 2^{11} = 2048 \rightarrow 512,\ 1024,\ 2048$

n	11	10	9	8	7	6	5	4	3	2	1	0
2^n	2048	1024	512	256	128	64	32	16	8	4	2	1

$110111_2 = 32 + 16 + 4 + 2 + 1 = 55$

2. Converting from base 10 to base 2: If the number is even, write a 0 on the leftmost place of the answer. If the number is odd, subtract 1 and write a 1 on the leftmost place of the answer. Divide the number by 2. Repeat if number does not equal 0.

3. Converting from base 2 to base 8 and vice versa using the **Rule of 3**: Every 3 bits on the binary number should equal every 1 bit on the octal number.

4. Conververting from base 2 to base 16 and vice versa using the **Rule of 4**: Every 4 bits on the binary number should equal every 1 bit on the hex number.

5. Useful binary formula:
$$2^n - 1 = 2^{n-1} + 2^{n-2} + \ ...+ 2^1 + 2^0 = \sum_{i=0}^{n-1} 2^i$$
$$8 - 1 = 1\ 000_2 - 1 = 111_2 \ \rightarrow\ 111_2 + 1 = 1\ 000_2$$
$$256 - 1 = 100\ 000\ 000_2 - 1 = 11\ 111\ 111_2 \ \rightarrow\ 11\ 111\ 111_2 + 1 = 100\ 000\ 000_2$$

6. $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$

7. $a \geq b \rightarrow a\%b = remainder\ when\ a \div b$
   $a < b \rightarrow a\%b = a$
   Sign depends on the the sign of the left number only.

Order of Precedence

P.U.M.A.S. R.E.B.eL. T.A.

Operators	**Precedence**
postfix	*expr*++ *expr*--
unary	++*expr* --*expr* +*expr* -*expr* ~ !
multiplicative	* / %
additive	**+ -**
shift	<< >> >>>
relational	< > <= >= instanceof
equality	== !=
bitwise AND	&
bitwise exclusive OR	^
bitwise inclusive OR	|
logical AND	&&
logical inclusive OR	||
ternary	? :
assignment	= += -= *= /= %= &= ^= |= <<= >>= >>>=

User-Defined Classes
Description: constructors, methods, instance variables, modifiers, overloading, overriding, ***final*** *local variables,* ***static final*** *class variables,* ***static*** *methods,* ***static*** *non-****final*** *variables*

1. Constructors
   a. Invoked to create objects.

    b. Have no return type and must use the name of the class.
    c. Cannot have the same number and type of arguments as another constructor.
    d. If no constructor is provided, the compiler provides a no-argument constructor that calls the constructor of the superclass.
        i. If the object has no explicit superclass, the superclass is implicitly **Object**.
    e. Can be public, protected, or private.
        i. If a class has children, there must be either no no-argument constructors or a visible no-argument constructor.
        ii. Private constructors can only be used inside the class.

2. Methods
    a. Declarations have six components:
        i. Modifiers such as public, private, or protected.
        ii. Return type such as void, primitive type, object type, or generic.
        iii. Method name.
            1. Case-sensitive and can only contain letters, numbers, dollar signs, and underscores. No spaces or reserved words.
        iv. Parameter list.
            1. A formal parameter is the identifier used in a method to stand for the value that is passed into the method by a caller.
            2. An actual parameter/argument is the actual value that is passed into the method by a called.
        v. Exception list.
        vi. Method body.
    b. Return type, method name, parameter list, and method body are necessary.
    c. Signature is composed of method name and parameter types.
    d. No two methods can have the same signature.
    e. Mutator methods set a variable and do not return a value.
    f. Accessor methods return a value.
3. Types of Variables
    a. Member variables in a class are called **fields**.
    b. Variables in a method of block of code are called **local variables**.
    c. Variables in method declarations are called **parameters**.
4. Instance Variables
    a. Objects store their individual states in **non-static** fields or instance variables.
    b. Instance variables are unique to each instance of the class.
5. Modifiers
    a. Visibility
        i. Public: Class, Package, Subclass, World
        ii. Protected: Class, Package, Subclass

        iii. No Modifier/Package-Private: Class, Package
        iv. Private: Class

6. Overloading
    a. An overloaded method is a method that has the same name as another method but different parameter types.
    b. Overloading can also apply to constructors.
7. Overriding
    a. An instance method in a subclass with the same signature and return type as an instance method in the superclass **overrides** the superclass's method.
8. Final Local Variables
    a. A method stores its state in local variables, which are only visible inside the method.
    b. The **final** modifier means that the variable cannot change from its initial value.
    c. All local variables must be instantiated with an explicit value to be used.
9. Static Final Class Variables
    a. Referred to as constants.
    b. Cannot be changed from its initial value and can be invoked by the class name or an instance of the class.
    c. Must be instantiated with an explicit value.
10. Static Methods
    a. Referred to as class methods.
    b. Can be invoked by the class name or an instance of the class.
    c. Instance methods can access everything that is visible.
    d. Class methods can only access **static** variables and methods that are visible. They must use an object reference to access instance variables and instance methods.
    e. Class methods cannot use the **this** keyword.
11. Static Non-final Variables
    a. Referred to as class variables.
    b. Any field declared with the static modifier.
    c. Regardless of how many instantiations of the class exist, only one copy of the variable exists.
    d. If no initial value is given, then the default value is used.

Constructors and Initialization of Static Variables, Default Initialization of Instance Variables

1. Static Variables
    a. Static non-final variables **do not** need an explicit value and can be instantiated in a static block. Static non-final variables can be instantiated in a constructor.

   b. Static final variables **do** need an explicit value or can be instantiated in a static block. Static final variables cannot be instantiated in a constructor.
2. Instance Variables
   a. Non-final instance variables **do not** need an explicit value and cannot be instantiated in a static block. Non-final instance variables can be instantiated in a constructor.
   b. Final instance variables **do** need an explicit value or can be instantiated in a constructor. Final instance variables cannot be instantiated in a static block.
3. Default Initialization

byte, short, int, long	0
float, double	0.0
char	null character w/ ASCII value of 0
boolean	false
any object including String	null

Concepts of Inheritance, Abstract Classes, Interfaces, and Polymorphism

1. Inheritance
   a. A class can extend another class, become a subclass of that class, and inherit visible fields and methods of a superclass.
   b. If no superclass is stated, default superclass is Object.
   c. The subclass inherits public and protected members of the superclass.
   d. Overriding and Hiding
      i. Declaring a field in the subclass with the same name of a field in the superclass "hides" the field in the superclass.
      ii. Writing a new instance method in the subclass with the same signature as a method in the superclass "overrides" the method in the superclass.
      iii. An overridden method in the child class cannot lower the visibility of the method in the parent class.
      iv. Writing a new static method in the subclass with the same signature as a method in the superclass "hides" the method in the superclass.
   e. A constructor in a subclass can call a constructor in the superclass using the keyword **super**. This must be the first line of code. If no call to a constructor of

the superclass is made, a call to the no-argument constructor of the superclass is called by default.

   f. A subclass can only have one superclass.
   g. A subclass can access a method in the superclass by using **super.method(args)**.
   h. If a subclass has no constructors, the no-argument constructor from the superclass is used.
      i. If the superclass doesn't have a no-argument constructor, a compile error will occur.
   i. A parent may be constructed as a child:

      Parent par = new Child();

      i. This is implicit casting.
   j. A child can only be constructed as a parent with an explicit cast:

      Child child = (Child) par;

2. Abstract Classes
   a. A superclass with at least one abstract method must be an abstract class.
   b. An class can be declared abstract without having an abstract method.
   c. Implementation for the abstract method is not provided, but must be provided in every subclass.
   d. Can have private instance variables, constructors, and concrete methods as well as abstract methods.
   e. Cannot be instantiated.
   f. An abstract class that implements an interface does not need to provide implementation for every method in the interface because those methods are also abstract.
   g. May have static fields and static methods.
   h. Can be used as a reference data type.
   i. <Subclass> is a <Superclass>.
3. Interfaces
   a. Classes can implement multiple interfaces.
   b. An interface is a group of method headers with no implementation.
   c. Methods are always abstract whether or not they are explicitly labeled as such.
   d. Contains no constructors.
   e. Contains only constants or static final fields.
   f. Cannot be instantiated.
   g. Only static methods and default methods can have implementation.
   h. Can extend other interfaces.
   i. Can be used as a reference data type.
   j. Classes that implement an interface must implement every method without implementation in the interface.

    k. <Subclass> is a <Interface>.
4. Polymorphism
    a. A method that has been overridden in at least one subclass is said to be polymorphic.
    b. A reference can only access methods and variables in the class of its type, not the object that the reference refers to.
    c. Example:

        A a = new B(); //B is a child class of A.
        a.doSomething();

        - Only allowed if class A has the method doSomething().
        - If class A has the method doSomething() and class B has the method doSomething(), the method in class B is used.
        - If class A has the method doSomething() and class B doesn't, then the method in class A is used.
        - If class B has the method doSomething() and class A doesn't, a compile-time error occurs.

Keywords

Description: null, this, super, super.method(args), super(args), this.var, this.method(args), this(args), instanceof

1. Keywords may not be used as identifiers. "true", "false", and "null" are literals, which also may not be used as identifiers.
2. Null
    a. Default value for all reference types is null.
    b. Cannot be a type or identifier.
3. This
    a. Reference to the current instance of an object.
    b. Refer to any member of the current object from within an instance method or a constructor by using **this** such as **this.var** and **this.method(args)**.
4. Super
    a. Reference to the superclass of a class.
    b. Refer to any public or protected member of the superclass from within the subclass by using **super**.
    c. A constructor of a subclass can invoke a constructor of its superclass by using **super** such as **super(args)**. This must be the first line of the constructor.
        i. If the subclass does not explicitly invoke a constructor of its superclass, the no-argument constructor of the superclass is invoked by default.

   d. If a method overrides one of its superclass's methods, invoke the overridden method through the use of the keyword **super** such as **super.method(args)**.

5. Instanceof
   a. The instanceof operator compares an object to a specified type.
   b. Returns true if a reference refers to an instance of a class, an instance of a subclass, or an instance of a class that implements a particular interface.
   c. Comparing an object using **instanceof** to null will cause a syntax error.
      i. Use "==" to compare an object to null.

Errors and Exceptions

1. Exceptions
   a. Exceptions can be handled in a try-catch block.
   b. If an exception occurs in Code 1, Code 1 will stop and the first appropriate catch block will execute, which in this case might be Code 2. Code 3 will run no matter what.

```
try {
        //Code 1
} catch (ExceptionType e) {
        //Code 2
} finally {
        //Code 3
}
```

3. Errors
   a. Compile-time
      i. Includes syntax and type checking errors.
      ii. Type checking errors are caused by the keyword **instanceof**.
      iii. Anything non-static referred to in a static context.
      iv. Variable that is used but not initialized.
      v. Missing return statements.
   b. Runtime
      i. Occurs after a code is compiled meaning that it is written in the correct Java syntax.
      ii. Includes Index Out Of Range, Null Pointer, Arithmetic (divide by 0), Illegal Format, Number Format, Out of Memory Class Cast Exceptions

Conversion to Supertypes and (Subtype) Casts

1. Implicit Casting: A parent reference can refer to a child object without an explicit cast.
2. Explicit Casting: A child reference can refer to a parent object only with an explicit cast.
    a. Casting an object to a reference is allowed if the object type and reference type are in the same hierarchy. For example, a string cannot be casted as an integer object.

Comparison of Reference Types

Description: equals(), ==, !=, Comparable.compareTo(), autoboxing, unboxing

1. Primitives
    a. Comparisons use the symbols “==”, “!=”, “<=”, and “>=”.
2. Reference Types
    a. Comparisons use the method “equals()” and “compareTo()”.
    b. Common concept:

```
String a = “hi”;
String b = “hi”;
System.out.println(a == b); //prints false
System.out.println(a.equals(b)); //prints true
```

    c. *object1.compareTo(object2)* will return a positive number if object 1 > object 2.
    d. *object1.compareTo(object2)* will return 0 if object 1 = object 2.
    e. *object1.compareTo(object2)* will return a negative number if object 1 < object 2.
        i. For strings, values are returned lexicographically. When comparing strings, subtract the ASCII values of the first differing character. If there are no differing characters, subtract the lengths. It is always *object1* subtracted by *object 2*.

```
System.out.println(“C”.compareTo(“A”)); //prints 2
System.out.println(“Goat”.compareTo(“Go”)); //prints 2
```

    f. When reference types are compared using “==”, the addresses are compared.

```
int a = 0;
Integer b = 0;
Integer c = new Integer(0);
Integer d = new Integer(0);

System.out.println(a == b); //true - autoboxing
System.out.println(a == c); //true - autoboxing
System.out.println(b == c); //false - autoboxed object does not equal
//instance object
System.out.println(c == d); //false - references point to different
//instances
```

```
c = d;

System.out.println(c == d); //true - references point to the same
//instance

c++;
d++;

System.out.println(c == d); //true - both are autoboxed objects
```

3. Autoboxing and Unboxing
   a. Autoboxing and unboxing is the automatic conversion the compiler makes between primitive types and their corresponding wrapper classes.
   b. For example, an int can be used as an Integer object without an explicit cast.
   c. Used when:
      i. a primitive is passed as a parameter to a method that expects an object of the corresponding wrapper class.
      ii. a primitive is assigned to a variable of the corresponding wrapper class.
      iii. an object of the wrapper class is passed as a parameter to a method that expects an object of the corresponding primitive.
      iv. an object of the wrapper class is assigned to a variable of the corresponding primitive.
   d. When a variable is passed as a parameter without a cast:
      i. Any numerical primitive type or wrapper class can be passed as any numerical primitive type of a larger or equal memory size.
      ii. Only the corresponding primitive can be passed as an object of the corresponding wrapper class.
      iii. Only Characters and chars can be passed as a Character and char.

Primitive Types, Casting of Primitives

Description: int, double, boolean, short, long, byte, char, float

Data Type	Storage Type	Memory Size	Max Value	Min Value
byte	integer	8 bit	$2^7 - 1 = 127$	$-2^7 = -128$
short	integer	16 bit	$2^{15} - 1 = 32767$	$-2^{16} = -32768$
int	integer	32 bit	$2^{31} - 1$	-2^{31}
long	integer	64 bit	$2^{63} - 1$	-2^{63}
float	floating-point	32 bit	N/A	N/A

double	floating-point	64 bit	N/A	N/A
char	character	16 bit	0	$2^{16} - 1 = 65535$
boolean	boolean	1 bit	true	false

1. Important ASCII Values:
   32 = SPACE
   48 = ‘0’
   65 = ‘A’
   97 = ‘a’
2. Floats have 23 bits of significand, 8 bits of exponent, and 1 sign bit. Floats must end in “f”.
3. Doubles have 52 bits of significand, 11 bits of exponent, and 1 sign bit.
4. The storage precision limit for a double value in effective decimal places is 15, while the float is only 7 places.
5. Hexadecimal constants start with “0x”.
6. Octal constants start with “0”.
7. Binary constants start with “0b”.
8. Casting
   a. For primitives, variables can be assigned to a variable of a smaller or equal memory size without a cast.

      ```
      int a = 17;
      long b = a;
      double c = b;
      a = (int) b;
      b = c;
      ```

   b. Floating-point data types never round when becoming integer data types. Instead, they are truncated.
   c. Floating-point data types will always have at least one place after the decimal place listed.

Arrays, including Arrays of Arrays and Initialization of Named Arrays

1. The variable that points to an array is a reference. If two references equal and reference the same array, then the changes to the array using one reference is the same change that occurs to the array of the other reference.

2. The declaration of an array includes the array's type, brackets indicating that the variable is an array, and the array's name.
   ```
   float[] array1;
   int array2[];
   String [] array3;
   ```

2. Initialize an array with the "new" operator. Must include size. Each index initially contains a default value.
   ```
   array = new String[10]; //creates a String array of size 10 with each index
   //containing null
   ```

3. An object array can contain instances of that object and its subclasses.
   ```
   Object[] array = new String[10];
   ```

4. Use braces to instantiate an array with custom values.
   ```
   Object[] array = {"String", 4, 12.0, 'c'};
   ```

5. The declaration of an array of an array includes the type of each inner array, two brackets, and the array's name.
   ```
   float[][] array1;
   int array2[][];
   String [] [] array3;
   ```

6. Use braces to instantiate an array of arrays with custom values.
   ```
   Object[][] array = {{0, 1.0}, {"String"}};
   ```

Arithmetic Operators and String Concatenation

*Description: +, -, *, /, %, ++, --, dividing by 0.0 and -0.0, NaN, positive and negative infinity, wrap around/overflow*

1. Once a string is added in an expression, added numerical data types are immediately strings. Arithmetic can still occur except for addition, but not with the string.
2. When an integer data type is divided by integer 0, an arithmetic exception occurs.
3. When a floating-point data type is divided by 0, 0.0, or -0.0:
   a. Answer is NaN if the numerator is 0, 0.0, or -0.0.
   b. Answer is Infinity if numerator has the same sign as the denominator and the numerator isn't 0, 0.0, or -0.0.
   c. Answer is -Infinity if numerator has the opposite sign as the denominator and the numerator isn't 0, 0.0, or -0.0.

4. Wrap around: when a primitive contains its max value and is incremented by 1, the primitive becomes the min value. When a primitive contains its min value and is decremented by 1, the primitive becomes the max value.

Using the Values of ++, -- Expressions in Other Expressions

1. Pre-increments and post-increments change the value stored in the variable even when used in an expression.
2. Pre-increments store the new value before the expression is evaluated.
3. Post-increments store the new value after the expression is evaluated.
4. **a+++b** is parsed as **(a++)+b**.
5. When evaluating expressions, use the java order of precedence and order of operations.
6. An expression with multiple primitives will become the data type with the largest memory size. Float-point data types are favored over integer data types.

Assignment Operators

*Description: =, +=, -=, *=, /=, %=*

1. The simple assignment operator "=" assigns the value on its right to the operand on its left.
2. Autocast is provided.
3. $x \mathrel{+}= a; \ \leftrightarrow \ x = x + a;$
4. $x \mathrel{-}= a; \ \leftrightarrow \ x = x - a;$
5. $x \mathrel{*}= a; \ \leftrightarrow \ x = x * a;$
6. $x \mathrel{/}= a; \ \leftrightarrow \ x = x / a;$
7. $x \mathrel{\%}= a; \ \leftrightarrow \ x = x \% a;$

Boolean Expressions and Operators including Short-Circuit Evaluation

Description: ==, !=, <, <=, >, >=, &&, ||, !, &, |, ^

1. Equality operators ("==", "!=") compare booleans.
2. Equality and relational operators ("==", "!=", "<" , "<=", ">", ">=") compare numerical data types.
3. The boolean && operator and boolean & operator perform a boolean AND operation.
4. The boolean || operator and boolean | operator perform a boolean OR operation.
5. The boolean ! operator performs a boolean NOT operation.
6. The boolean ^ operator performs a boolean XOR operation.
7. Short-circuit evaluators may not evaluate the second condition whereas standard evaluators will always evaluate the second condition.

a. When two conditions are joined with "||" and the first condition is true, the entire condition is true and the second condition is not evaluated.
b. When two conditions are joined with "&&" and the first condition is false, the entire condition is false and the second condition is not evaluated.
c. Often used to prevent errors that may be caused by the second condition.

```
int a = 1;
int b = 0;
if(b != 0 && a / b > 0) //no error because of short-circuiting
        System.out.println("Signs equal!");
```

Bitwise Operators

Description: <<, >>, >>>, &, ~, |, ^, >>=, <<=, &=, |=, ^=

1. Signed left shift "<<" shifts a bit pattern to the left with 0s to pad the right.
   $a > 0 \ \& \ n > 0 \rightarrow a << n = a * 2^n$
2. Signed right shift ">>" shifts a bit pattern to the right with 0s to pad the left if the leftmost bit is 0 or 1s to pad the left if the leftmost bit is 1.
   $a > 0 \ \& \ n > 0 \rightarrow a >> n = a \,/\, 2^n$
3. Unsigned right shift ">>>" shifts a bit pattern to the right with 0s to pad the left.
4. The bitwise & operator performs a bitwise AND operation.
   1 1101 & 1110 = 1100
5. The bitwise | operator performs a bitwise OR operation.
   1 1101 | 1110 = 1 1111
6. The bitwise ~ operator inverts every bit.
   ~ 11010 = 101
7. The bitwise ^ operator performs a bitwise XOR operation.
   1 1101 $\wedge$ 1110 = 1 0011
8. Bitwise assignment operators have the same function and format as standard assignment operators.

Branching

Description: if, if/else, ?:, switch, break

1. Code 1 and Code 2 will only execute if boolean expression is true.

```
if(boolean expression) {
        //Code 1
        //Code 2
}
```

2. Code 1 will only execute if boolean expression is true. Don't be fooled! Code 2 will always execute.

```
if(boolean expression)
        //Code 1
        //Code 2
```

3. Code 1 will execute if boolean expression is true. Code 2 will execute if boolean expression if false.

```
if(boolean expression) {
        //Code 1
} else {
        //Code 2
}

OR

if(boolean expression)
        //Code 1
else
        //Code 2
```

   This will cause a compile-time error:

```
if(boolean expression)
        //Code 1
        //Code 2
else
        //Code 3
```

4. Expression 1 is used if boolean expression is true. Expression 2 will execute if boolean expression is false.

```
(boolean expression)?Expression 1:Expression 2
```

5. A switch is a group of cases. Each case corresponds to a value and contains code. The case that will execute is determined by the value of a variable. If the value of var doesn't match with any case, the default case is used. Any case statement that doesn't end in **break** will execute all code after the case until a **break** is found or the end of the switch statement is reached. For example, if the value of var equals value3, then Code 3, Code 4, and Code 5 will execute. Var can only be a primitive, class that wraps a primitive, enumerated type, or string.

```
switch(var) {
        case value1: Code 1
                break;
        case value2: Code 2
                break;
        case value3: Code 3
        case value4: Code 4
        default: Code 5
                break;
}
```

Looping

Description: while, for, ***enhanced*** *for, do/while, break, continue, return*

1. If a and b are integers, then the number of integers the range $[a,\ b]$ is $b-a+1$.
2. To trace a loop, kept track of the value of every variable as you read the code line by line.
3. To calculate how many times a for loop will iterate, write the range in the form $[a,\ b]$ and subtract a and b by the same amount such that a is divisible by the increment. Round b down to the next number that is divisible by the increment. Then, divide a and b by the increment and apply the formula $b-a+1$.
4. While boolean expression is true, Code 1 is repeatedly execute. If boolean expression is initially false, then Code 1 will not run at all.

```
while(boolean expression) {
        //Code 1
}
```

2. Step 1 → Step 2 → Go to Step 3 if boolean is true. → Step 3 → Step 4 → Back to Step 2

```
for( Step 1 initial; Step 2 boolean; Step 4 increment) {
        Step 3 code
}
```

3. Loop will iterate through an array or a collection without creating an iterator or a counter.

```
for(DataType varName : Collection) {
        //Code
}
```

4. Do-while loops will always run once and continue running while the boolean is true.

```
do {
        //Code
```

} while (*boolean*);

5. The **break** statement inside a loop will exit the loop.
6. The **continue** statement inside a loop will skip the current the iteration and go to the next iteration.
7. The **return** statement in general will stop the code immediately and may or may not return a value.

Console Output

Description: System.out.print(), println(), printf(), %f, %d, %s, %b, escape sequences, \n, \\, \"

1. System.out.print(*data*) writes the data to the console.
    a. If *data* is null, "null" is printed.
2. System.out.println(*data*) writes the data to the console and adds a newline character.
    a. If *data* is null, "null" is printed.
3. System.out.printf(*format*, *args*) writes the formatted string with the arguments referenced by the format.
4. String Formatting
    a. "%f" references a floating-point argument.
    b. "%d" and "%i" reference an integer argument.
    c. "%s" references a String argument and can be used for any primitive.
    d. "%b" references a **boolean**, is false for null and false boolean, and is true for any other primitive.
    e. "%o" references an octal integer.
    f. "%x" references a hex integer.
    g. "%u" references an unsigned integer.
    h. "%e" references an exponential floating-point argument.
    i. Formats are default right-justified and padded with spaces.
    j. Options listed are in order from left to right.
    k. Left Justify
        i. A minus sign after the percent sign indicates left-justify.
    l. Show Positive Sign
        i. A plus sign after the percent sign will show the plus sign for positive numbers and 0.
    m. Parentheses around Negative Number
        i. An open parenthesis after the percent sign will place parentheses around a negative number.
    n. Zero-fill Option
        i. A zero after the percent sign indicates padding with zeros.

    o. Controlling the Width
        i. Integer between the percent sign and letter indicates width.
        ii. If the integer is less than the required width, the required width is used.
        iii. If the integer is more than the required width, the integer is used.
    p. Positions after the Decimal Place
        i. A period and integer indicates the number the places after the decimal place should be displayed.
        ii. Rounding occurs.
5. A character preceded by an backslash is an escape sequence, which is needed for special characters.
    a. Escape sequence is needed for double quotes and backslash in strings.
    b. Escape sequence is needed for single quote and backslash in chars.
    c. \t is a tab.
    d. \n is a newline.
    e. \b is a backspace.
    f. \r is a carriage return. Text after the carriage return rewrites text starting at the beginning of the line.

Parsing

Description: String.split(), Integer.parseInt(), Double.parseDouble()

1. String.split(*regex*) returns an array of strings computed by splitting the string around matches of the given regular expression.
    a. Trailing empty strings are not included.
    b. If the first match begins the string, an empty string is included.
    c. When inserting a special character, two backslashes or braces are needed.
2. Integer.parseInt(*string, radix*) returns the int represented by the string in the specified radix.
    a. Converts an integer represented by the string in the specified base **to base 10**.
3. Integer.parseInt(*string*) returns the int represented by the string.
4. Double.parseDouble(*string*) returns the double represented by the string.

Pattern Class, Regex, Matches

*Description: ., +, *, \d, \D, \s, \S, \w, \W, [abc], [^abc], [a-zA-z]*

1. A regular expression is a template that certain character sequences can match.
2. The method Pattern.matches(*regex*, *input*) determines if the input matches the regular expression.
3. In a regular expression,

a. “.” represents any character.
b. “+” represents one or more of the preceding item.
c. “*” represents zero or more of the preceding item.
d. “^” inside a character class (“[]”) represents a negation.
e. “\d” represents a digit.
f. “\D” represents a non-digit.
g. “\s” represents a whitespace character.
h. “\S” represents a non-whitespace character.
i. “\w” represents a word character. ([a-zA-Z_0-9])
j. “\W” represents a non-word character.
k. “[abc]” represents a, b, or c.
l. “[^abc]” represents not a, b, and c.
m. “[a-zA-Z]” represents all lower and upper case characters.

Java Standard Library

Description: String, Integer, Double, Character, Math, Object, Comparable, Scanner, Random, Arrays

1. The following list of methods in each class will only contain non-descriptive methods that might be tested. For a comprehensive list, refer to the official Java API.
2. When a fromIndex and toIndex are parameters, fromIndex is inclusive and toIndex is exclusive by convention.
3. String
    - Strings are immutable, so strings cannot be changed by the method itself.
    - int indexOf(String str)
        - Returns the index within this string of the first occurrence of the specified substring.
    - int indexOf(String str, int fromIndex)
        - Returns the index within this string of the first occurrence of the specified substring, starting at the specified index.
    - int lastIndexOf(String str)
        - Returns the index within this string of the last occurrence of the specified substring.
    - int lastIndexOf(String str, int fromIndex)
        - Returns the index within this string of the last occurrence of the specified substring, searching backward starting at the specified index.
    - boolean matches(String regex)
        - Tells whether or not this string matches the given regular expression.

- boolean regionMatches(boolean ignoreCase, int toffset, String other, int ooffset, int len)
    - Returns true if these substrings represent identical character sequences ignoring the case if necessary.
- boolean regionMatches(int toffset, String other, int ooffset, int len)
    - Returns true if these substrings represent identical character sequences.
- String replace(String target, String replacement)
    - Replaces each substring of this string that matches the literal target sequence with the specified literal replacement sequence.
- String replaceAll(String regex, String replacement)
    - Replaces each substring of this string that matches the given regular expression with the given replacement.
- String replaceFirst(String regex, String replacement)
    - Replaces the first substring of this string that matches the given regular expression with the given replacement.
- String[] split(String regex, int limit)
    - Splits this string around matches of the given regular expression with a specified maximum amount of splits.
- boolean startsWith(String prefix, int toffset)
    - Tests if the substring of this string beginning at the specified index starts with the specified prefix.
- String substring(int beginIndex)
    - Returns a new string that is a substring of this string.

4. Integer
    - static int parseInt(String s, int radix)
        - Parses the string argument as a signed integer in the radix specified by the second argument.
    - static String toString(int i, int radix)
        - Returns a string representation of the first argument in the radix specified by the second argument.
    - static Integer valueOf(String s, int radix)
        - Returns an Integer object holding the value extracted from the specified String when parsed with the radix given by the second argument.
5. Double
    - static double parseDouble(String s)
        - Returns a new double initialized to the value represented by the specified String, as performed by the valueOf method of class Double.
6. Character
    - static int digit(char ch, int radix)

    - Returns the numeric value of the character ch in the specified radix.
- static char forDigit(int digit, int radix)
    - Determines the character representation for a specific digit in the specified radix.

7. Math

- static int abs(int a)
    - Returns the absolute value of an int value.
- static double abs(double a)
    - Returns the absolute value of a double value.
- static double cbrt(double a)
    - Returns the cube root of a double value.
- static double ceil(double a)
    - Returns the smallest double value the is greater than or equal to the argument and is equal to a mathematical integer.
- static double exp(double a)
    - Returns Euler's number e raised to the power of a double value.
- static double expm1(double x)
    - Returns $e^x - 1$.
- static double floor(double a)
    - Returns the largest double value that is less than or equal to the argument and is equal to a mathematical integer.
- static double hypot(double x, double y)
    - Returns $\sqrt{x^2 + y^2}$.
- static double log(double a)
    - Returns the natural logarithm of a double value.
- static double log10(double a)
    - Returns the base 10 logarithm of a double value.
- static double log1p(double a)
    - Returns the natural logarithm of the sum of the argument and 1.
- static int max(int a, int b)
    - Returns the greater of the two values.
- static double nextAfter(double start, double direction)
    - Returns the floating-point number adjacent to the first argument in the direction of the second argument.
- static double pow(double a, double b)
    - Returns the value of the first argument raised to the power of the second argument.
- static double random()

        - Returns the double value with a positive sign, greater than or equal to 0.0 and less than 1.0.
    - static double rint(double a)
        - Returns the double value that is closest in value to the argument and is equal to a mathematical integer.
    - static long round(double a)
        - Returns the closest long to the argument with ties rounding up.
8. Object
    - boolean equals(Object obj)
        - Indicates whether some other object is “equal to” this one.
    - String toString()
        - Returns a string representation of the object.
        - *ClassName@HashCode*
9. Comparable
    - int compareTo(T o)
        - Compare this object with the specified object for order.
10. Scanner
    - boolean hasNext()
        - Returns true if this scanner has another token in its input.
    - boolean hasNext(String pattern)
        - Returns true if the next token matches the pattern.
    - boolean hasNext*[PrimitiveType]*()
        - Returns true if the next token has the specified primitive type.
    - String next()
        - Returns the next token.
    - String next(String pattern)
        - Returns the next token if it matches the pattern.
    - boolean nextBoolean()
    - double nextDouble()
    - int nextInt()
    - int nextInt(int radix)
        - Returns the next token interpreted with the specified radix.
    - String nextLine()
    - short nextShort()
11. Random
    - double nextDouble()
        - Returns a random double from 0.0 inclusive to 1.0 exclusive.
    - int nextInt(int n)
        - Returns a random int from 0 inclusive to n exclusive.

12. Arrays
    - static int binarySearch(int[] a, int fromIndex, int toIndex, int key)
        - Searches a range of the specified array of ints for the specified value using the binary search algorithm.
        - The max number of comparisons for an array of x elements such that $2^{n-1} < x \leq 2^n$ is n.
        - If key is not in a, returns $-1 - insertion\ point$.
    - static int[] copyOf(int[] original, int newLength)
        - Copies the specified array, truncating or padding with zeros.
    - static void fill(int[] a, int fromIndex, int toIndex, int val)
        - Assigns the specified int value to each element of the specified range of the specified array of ints.
    - static String toString(int[] a)
        - Returns a string representation of the contents of the specified array.

Generic Collections

Description: Collection, List, Set, Map, Stack, Queue, PriorityQueue, ArrayList, LinkedList, HashSet, TreeSet, HashMap, TreeMap

1. An integer as the parameter in a constructor initializes the internal capacity, not the size. The internal capacity changes automatically when needed.
2. A collection as the parameter in a constructor contains the elements that are added to the new collection.
3. When instantiating a collection, a type is needed inside the diamond braces on the left but not on the right.
4. Collection
    - Root interface in the collection hierarchy.
    - Iterator<E> iterator()
        - Returns an iterator for the collection.
        - Iterators can only call remove() once per next().
        - Iterators operator the same way as a typing cursor.
    - int size()
5. List
    - Interface
    - Indexed list of elements.
    - void add(int index, E element)
        - Shifts every element on and after index over and adds the element at the specified index.
    - E set(int index, E element)

        - Replaces the element at the specified position in this list with the specified element.

6. Set
    - Interface
    - Collection that contains no duplicates.
7. Map
    - Interface
    - Collection that maps keys to values.
    - No duplicate keys.
        - Overrides previous key if duplicate key is inserted.
    - V get(Object key)
        - Returns the value to which the specified key is mapped or null if this map contains no mapping for the key.
    - V put(K key, V value)
        - Returns the previous value associated with the key or null.
8. Stack
    - Class
    - Top element is at index 1.
    - Prints from bottom to top.
    - Last-in-first-out stack of objects.
    - E peek()
        - Returns the top of the stack without removing it.
    - E pop()
        - Removes the top of the stack and returns the object.
    - E push(E item)
        - Adds item to the top of the stack.
    - int search(Object o)
        - Returns the 1-based position where an object is on this stack.
9. Queue
    - Interface
    - First-in-first-out queue of objects.

	Throws Exception	Returns Special Value
Insert	add(e)	offer(e)
Remove	remove()	poll()
Examine	element()	peek()

10. PriorityQueue

- Class
- Implements a min heap.
- The toString() representation prints the nodes level by level.
- To print the nodes from least to greatest, you must print the nodes that are removed by the method poll() or remove().

11. ArrayList
    - Class
    - Resizable-array implementation of a list.
    - An arraylist is a random access data structure, where each element can be accessed directly and in constant time.
    - boolean addAll(int index, Collections<? extends E> c)
        - Inserts every element in the collection at a specified index.
12. LinkedList
    - Class
    - Doubly-linked list implementation of a list.
    - E element()/E peek()
        - Retrieves, but does not remove, the first element of this list.
    - boolean offer(E e)
        - Adds the specified element as the last element of this list.
    - E poll()/E pop()/E remove()
        - Retrieves and removes the first element of this list.
    - E push()
        - Adds the specified element as the first element of this list.
    - E remove(int index)
        - Retrieves and removes the element at the specified index.
        - Will be used when the argument is an int.
    - boolean remove(Object o)
        - Removes the first occurrence of the object if present.
13. HashSet
    - Class
    - Implements a hash table.
    - Each element is inserted based on its hash code, so order is not guaranteed.
    - Constant time performance for basic operations (add, remove, contains, and size).
14. TreeSet
    - Class
    - Guarantees order.
    - O(log n) time performance for basic operations (add, remove, and contains).
    - E ceiling(E e)/E higher(E e)

        - Returns the least element in this set greater than or equal to the given element, or null if there is no such element.
    - E floor(E e)/E lower(E e)
        - Returns the greatest element in this set less than or equal to the given element, or null if there is no such element.

15. HashMap
    - Class
    - Implements a hash table.
    - Each element is inserted based on its hash code, so order is not guaranteed.
    - Constant time performance for basic operations (get and put).
16. TreeMap
    - Class
    - Guarantees order.
    - O(log n) time performance for basic operations (containsKey, get, put, and remove).

Arrays.sort(), Collections.sort()

1. Arrays.sort(*array*) and Arrays.sort(*array, fromIndex, toIndex*) use a Dual-Pivot **Quicksort**.
    a. If *fromIndex* = *toIndex*, the range to be sorted is empty. No exception is thrown.
2. Collections.sort(*list*) uses a stable, adaptive, iterative **mergesort**.

Recursion

1. Write the equation for the initial value of the function.
2. If another unknown value of a function is needed, write the equation for the unknown value of the function a line underneath.
3. Repeat step 2 until an equation without an unknown is found.
4. Substitute back up the stack until the top line is reached.

Data Structures

Description: Stacks, Queues, Binary Trees, Linked Lists, Heaps, Hash Tables, Priority Queues, Graphs

1. Stacks
    - Last-in-first-out stack of objects.
    - Think of a stack of books.
2. Queues
    - First-in-first-out queue of objects.

    - Think of a lunch line.
3. Binary Trees
    - Each node has at most two children.
    - In a full binary tree, every node has two children except for the leaves.
    - In a complete binary tree, every level is complete except possibly the last level.
    - A binary tree with n levels can have at most $2^n - 1$ nodes.
    - To print in preorder, add a dot on the left of every node, start on the left side of the root, and trace around the tree until you reach the right side of the root. Print a node every time the line reaches the dot.
    - To print in inorder, add the dot on the bottom of every node and do the same.
    - To print in postorder, add the dot on the right of every node and do the same.
4. Linked Lists
    - A linked list is a sequential access data structure, where each element can be accessed only in particular order.
    - Insertion and deletion of a node is constant time because they only require changing references.
    - Accessing the element in a specified index is linear time performance.
5. Heaps
    - Can be a min heap or a max heap.
    - For a min heap, every parent must be less than its children.
    - For a max heap, every parent must be less than its children.
    - To retain this property, the heap must *heapify* every time an element is inserted or deleted.
    - When inserting elements, insert level by level and *heapify* after every insertion.
    - During a deletion, remove the root, use the last node as the new root, and *heapify*.
    - To *heapify*, each parent and child must follow the heap property. During insertion, the node that was added will swap with its parent until the heap property is satisfied. During deletion, the new root node will swap with its smallest child for a min heap or largest child for a max heap until the heap property is satisfied.
6. Hash Tables
    - A hash table is an array with a method of calculating a hash code for an object. The hash code determines the index in the array.
    - This allows for constant time performance for access, insertion, and deletion.
7. Priority Queues
    - Priority queues implement a min heap.
    - *See heaps.*
8. Graphs
    - Graphs are composed of nodes, which each contain a value and list of references to other nodes.

- *See graph theory.*

Sorts and Searches

Description: Selection, Insertion, Mergesort, Quicksort, Sequential Search, Binary Search

1. Selection Sort

```
array a = array to be sorted
for i = 0 to length of a - 1
    indexOfMin = i
    for j = i + 1 to length of a
        if a[j] < a[indexOfMin]
            indexOfMin= j
    swap a[i] and a[indexOfMin]
```

2. Insertion Sort

```
array a = array to be sorted
for i = 1 to length of a - 1
    j = i
    while j > 0 and a[j - 1] > a[j]
        swap a[j] and a[j - 1]
        j = j - 1
```

3. Mergesort

```
function mergesort(array a)
    if length of a <= 1
        return a
    middle = length of a / 2

    array left = new array of size (middle)
    array right = new array of size (length of a - middle)
    for each x in a before middle
        add x to left
    for each x in m after or equal to middle
        add x to right

    left = mergesort(left)
    right = mergesort(right)

    return merge(left, right)
```

```
function merge(array left, array right)
    array result = new array of size (length of a + length of b)
    i = 0
    l = 0
    r = 0
    while l < length of left and r < length of right
        if left[l] <= right[r]
            result[i++] = left[l++]
        else
            result[i++] = right[r++]
    while l < length of left
        result[i++] = left[l++]
    while r < length of right
        result[r++] = right[r++]
    return result
```

4. Quicksort

```
function quicksort(array a, int low, int high)
    if low < high
        p = partition(a, low, high)
        quicksort(a, low, p - 1)
        quicksort(a, p + 1, high)

function partition(array a, int low, int high)
    pivotIndex = low + (high - low) / 2
    pivotValue = a[pivotIndex]
    swap a[pivotIndex] = a[high]
    storeIndex = low
    for i = low to high
        if a[i] <= pivotValue
            swap a[i] and a[storeIndex]
            storeIndex = storeIndex + 1
    swap a[storeIndex] and a[high]
    return storeIndex
```

5. Sequential Search

```
array a = array to be searched
key = item to be found
```

```
for i = 0 to length of a
    if a[i] = key
        return i
return -1
```

6. Recursive Binary Search

```
function binarySearch(array a, int high, int low, item key)
    index i = low + (high - low) / 2
    if length of a = 0
        return -1
    else if a[i] = key
        return i
    else if a[i] > key
        return binarySearch(a, 0, i, key)
    else
        return binarySearch(a, i + 1, high, key)
```

7. Iterative Binary Search

```
array a = array to be searched
key = key to be found
low = 0
high = length of a
index = low + (high - low) / 2
while a[index] != key
    if a[index] > key
        high = index - 1
    else
        low = index + 1
    if low > high
        return -1
return index
```

Analysis of Algorithms

Description: informal comparison of running times, exact calculation of statement execution counts, Big-O notation, best case/worst case/average case time, space analysis

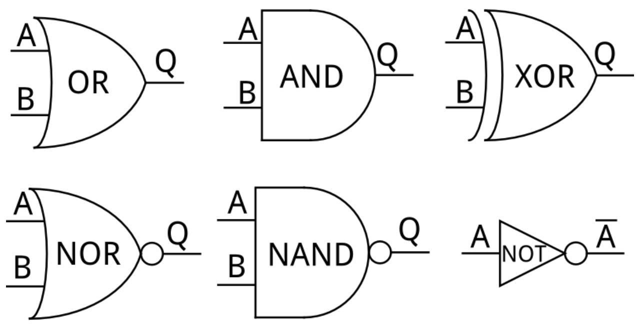
$O(1) < O(log\ n) < O(n) < O(n\ log\ n) < O(n^2)$

Data Structure	Average Time Complexity			
	Access	Search	Insertion	Deletion
Array	O(1)	O(n)	O(n)	O(n)
Linked List	O(n)	O(n)	O(1)	O(1)
Hash Table	-	O(1)	O(1)	O(1)
Stack	O(n)	O(n)	O(1)	O(1)
Binary Search Tree	O(log n)	O(log n)	O(log n)	O(log n)

Algorithm	Time Complexity		
	Best	Average	Worst
Quicksort	O(n log n)	O(n log n)	O(n^2)
Mergesort	O(n log n)	O(n log n)	O(n log n)
Heapsort	O(n log n)	O(n log n)	O(n log n)
Bubblesort	O(n)	O(n^2)	O(n^2)
Insertionsort	O(n)	O(n^2)	O(n^2)
Selectionsort	O(n^2)	O(n^2)	O(n^2)

1. $t_1 = time_1 \wedge t_2 = time_2 \wedge n_1 = data\ size_1 \wedge n_2 = data\ size_2$
   $\rightarrow \frac{t_1}{t_2} = O(\frac{n_1}{n_2})$
2. If an algorithm takes t_2 seconds to process n_2 elements and the algorithm's time complexity is O(n^x), how long does the algorithm take to process n_1 elements?
   $t_1 = t_2 \times (\frac{n_1}{n_2})^x$

Digital Electronics



- Be sure to include parentheses in the correct positions when translating digital electronic diagrams.

Two's Complement Binary Representation of Negative 8-bit Integers
Description: conversion both ways between base 10 and base 2

1. Leftmost bit is reserved for the sign.

   -1 = **1**111 1111
   -7 = **1**111 1001
   -31 = **1**110 0001
   -128 = **1**000 0000

   1 = **0**000 0001
   7 = **0**000 0111
   31 = **0**001 1111
   127 = **0**111 1111

2. Converting a Negative 8-bit Integer from Base 2 to Base 10: Flip every bit to the left of the rightmost 1. Convert the positive resulting number in base 2 to base 10 and multiply by -1.

   $1111\ 0011 \rightarrow 0000\ 1101 \rightarrow 13 \rightarrow -13$
   $1110\ 0100 \rightarrow 0001\ 1100 \rightarrow 28 \rightarrow -28$
   $1111\ 1111 \rightarrow 0000\ 0001 \rightarrow 1 \rightarrow -1$

3. Converting a Negative 8-bit Integer from Base 10 to Base 2: Multiply by -1 and subtract 1. Convert the positive resulting number in base 10 to base 2. Flip every bit.

   $-15 \rightarrow 14 \rightarrow 0000\ 1110 \rightarrow 1111\ 0001$
   $-21 \rightarrow 20 \rightarrow 0001\ 0100 \rightarrow 1110\ 1011$
   $-1 \rightarrow 0 \rightarrow 0000\ 0000 \rightarrow 1111\ 1111$

Polish Notation

Description: representation, analysis, and conversion of simple infix, prefix, and postfix expressions

1. Values should be kept in the same order when converting between notations.
2. For prefix notation, operators are before two operands.
3. For infix notation, operators are between two operands.
4. For postfix notation, operators are after two operands.
5. Place parentheses around every operator and its corresponding operands and place the operator in the correct position inside every parentheses to convert between notations.

$$A - B / (C + D) \rightarrow (A - (B / (C + D))) \rightarrow (-A(/B(+C\ D))) \rightarrow \ -A / B + CD$$

OR

$$A - B / (C + D) \rightarrow (A - (B / (C + D))) \rightarrow (A(B(C\ D+)/)-) \rightarrow \ ABCD + / -$$

Boolean Simplification using Generic Notation

*Description: A*B, A+B,* $\overline{A}$, $\overline{A * B}$, $\overline{A + B}$, $\overline{A \oplus B}$, *using truth tables and boolean identities to analyze and simplify Boolean expressions*

1. Distributive Law

   $A * (B + C) = A * B + A * C$

   $A + (B * C) = (A + B) * (A + C)$
2. Law of Union

   $A + 1 = 1$
3. Law of Intersection

   $A * 0 = 0$
4. Law of Absorption

   $A + (A * B) = A$
5. Identity Law

   $A * 1 = A$

   $A + 0 = A$
6. Law of Complement

   $A + \overline{A} = 1$

   $A * \overline{A} = 0$
7. DeMorgan's Law

   $\overline{A + B} = \overline{A} * \overline{B}$

   $\overline{A * B} = \overline{A} + \overline{B}$

   Be careful!

$$\overline{A+BC} = \overline{A} * \overline{BC} = \overline{A} * (\overline{B} + \overline{C})$$

8. Exclusive Or/Exclusive Nor

$$A \oplus B = A * \overline{B} + \overline{A} * B$$

$$\overline{A \oplus B} = A * B + \overline{A} * \overline{B}$$

9. Law of the Disappearing Opposite

$$A + \overline{A} * B = A + B$$

Bit String Flickering

1. Operators
   a. There are the four basic boolean operators NOT, AND, OR, and XOR.
   b. In addition, there are the CIRC and SHIFT operators: LCIRC, RCIRC, LSHIFT, and RSHIFT.
   c. LSHIFT/RSHIFT operators will always pad with zeros.
   d. LCIRC/RCIRC operators will circulate the bits.
   e. The asterisk symbol (*) is used to denote an ambiguous bit that can be either 0 or 1.
   f. When solving for a variable, use variables to represent every bit in the missing variable and solve for each individual bit variable.

Graph Theory

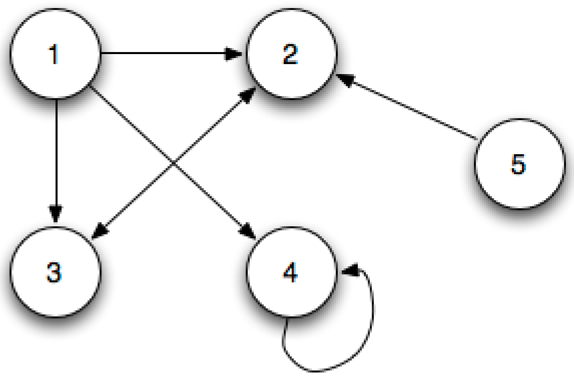
graph - a set of vertices together with a set of edges connecting the vertices
complete graph - a graph with **an edge connecting every pair** of distinct vertices
connected graph - a graph such that there **exists a path** between any two vertices
regular path - a sequence of edges that connects two vertices
simple path - a path with **no repeated** edges
minimal path - the shortest path from a vertex to another
cycle - a **simple** path that starts and ends at the same vertex
tree - a connected graph with no cycles
bipartite graph - a graph that can be partitioned into two sets such that every edge connects a vertex in one set to a vertex in another
planar graph - a graph such that none of its edges intersect
face - a region of the plane with no edges in its interior and whose boundary is a union of edges of the graph (the region on the outside of the graph is a face)
euler's formula:

$$V + F = E + 2$$

eulerian path - a path that contains **every edge** of the graph **exactly once**
eulerian cycle - a eulerian path that is also a cycle
hamiltonian path - a path that contains **every vertex** of the graph **exactly once**

hamiltonian cycle - a hamiltonian path except that the path starts and ends on the same vertex

adjacency matrix - a matrix in which each row represents a starting vertex and each column represents an ending vertex



adjacency matrix:

	1	2	3	4	5
1	0	1	1	1	0
2	0	0	1	0	0
3	0	1	0	0	0
4	0	0	0	1	0
5	0	1	0	0	0

To find the number of *n*-length paths from node A to node B, raise the matrix to the *nth* power and find the number on row A, column B. *Review multiplying matrices.*